

9. Development Environment

The DII COE imposes very few requirements on the process or tools developers use to design and implement software. The COE concentrates on the end product and how it will integrate in with the overall system. This approach provides the flexibility to allow developers to conform to their internal development process requirements. However, developers are expected to use good software engineering practices and development tools to ensure robust products. The purpose of this chapter is to suggest certain development practices that will reduce integration problems, and the impact of one segment on another.

Developers may select compilers, debuggers, linkers, editors, CASE tools, etc. that are most suitable for their development environment. The compilers and linkers selected must be compatible with the products supplied by the hardware vendors and must not require any special products for other developers to acquire in order to use the segments produced.

9.1 Coding Conventions

This section describes required coding standards for segments submitted to DISA, whether they are COE-component segments, or mission-application segments that are part of a DISA COE-based system. These standards are not intended to restrict software development, and for that reason the requirements given are brief.

There are two important points to keep in mind with respect to this chapter. First, the DII COE states requirements for the purpose of ensuring and preserving the integrity of the runtime environment. Therefore, the DII COE is mostly concerned with executables that are produced, and not the process used to create them. The COE relies upon other standards (e.g., MIL-STD 2167A, MIL-STD 495, ISO 9000) and practices levied by the cognizant program managers to ensure good programming practices and a quality product. However, certain standards are required because some of the segments produced contain APIs that developers will use to build other segments upon.

Secondly, the DII COE is programming-language neutral and does not stipulate what programming language to use to write segments. Such decisions are the prerogative of the cognizant program manager. The COE must support segments written in Ada, in support of DOD policy, and C, because of the use of COTS products, and therefore both are addressed in this chapter. Any statements in this chapter, or elsewhere in the *I&RTS*, which appear to state a preference for one language over another are unintentional.

Because most developers are using either C/C++ or Ada, COE-component segments that provide APIs shall be written in either C/C++ or Ada. Availability of APIs for both C and Ada is highly desirable, but will be driven by service and agency requirements. Consult with the DII COE Chief Engineer for availability of multi-language APIs, for requirements to produce multi-language APIs for a particular segment, or for support for languages other than C/C++ and Ada.

9.1.1 Language Independent Conventions

The following suggestions and requirements are language independent.

- Code delivered to DISA shall *not* be compiled with debug options enabled. If available, a utility such as the Unix `strip` command shall be run on executables to minimize the disk space required.
- Segments should use shared libraries where practical to reduce runtime memory requirements. Segments with public APIs implemented as shared libraries shall also be delivered as static libraries to make debugging easier for developers who need to use the APIs.
- Developers may use GUI (Graphical User Interface) tools to build interfaces, but developer's should select tools that are portable across platforms. Segments built with such tools shall use resource files for window behavior rather than embedded code,

and must not require any runtime licenses unless approved by the DII COE Chief Engineer for COE segments, or by the cognizant program manager for application segments.

- Developers should run all modules through a tool such as `lint` to detect potential coding errors prior to compiling.
- Developers should run all modules through commercially available tools to detect as many runtime errors as possible (e.g., “memory leaks”).
- Developers should periodically profile segments by using tools that do a runtime analysis of module performance (% CPU utilization, number of times a function is invoked, amount of time spent in a function, LAN loading analysis, etc.).
- Developers should create a test suite for automatically exercising the segment, especially inter-segment interfaces and APIs, and periodically run the tests to perform regression testing. A formal test plan should be created and submitted with the segment.
- Segments with public APIs shall be delivered with a test suite that covers all public APIs provided by the segment.
- Developers should use a tool such as `imake` for generating `makefiles` that are as portable as possible. If available, the POSIX.2 `make` utility should be used.
- Developers should use automated tools such as CVS, RCS, or other commercially available product to perform configuration management tasks. Segment developers are responsible for configuration control of their own products. The *I&RTS* does not proscribe a CM plan, but assumes the developer has one as part of good programming practices.
- Developers should periodically rebuild segments from scratch to ensure that all pieces, including data files, are under proper configuration management control.
- Developers should track problem reports in an automated database. This will simplify reporting known problems when the segment is submitted to the cognizant SSA.
- Shareware and freeware products should generally be avoided. Products such as the gnu software should be used with care because the licensing agreement may require distribution of source code and may thus have adverse impact on product releasability.
- Developers shall separate COTS products from mission-application software because the COTS software may already be available in the DII COE inventory.

9.1.2 Ada

Ada generally requires stipulating fewer requirements than other languages because the syntax and semantics of the language are designed to enforce good programming practices at the compiler level. For example, Ada enforces strong typing so that many common coding errors are caught at compile time.

Ada bindings in particular pose specific areas of concern.

- Developers should design software so that routines that require binding to other languages are isolated into a small number of easily separated modules. This will make maintenance of Ada bindings easier, and make it easier to identify segments that require long-term support for Ada bindings.
- Developers who create Ada bindings to other segments or COTS products within the COE should submit them with their segment so that other developers may reuse them.
- Developers who require Ada bindings to COTS products within the COE (e.g., Motif, DCE) should use commercially available bindings whenever they exist, and whenever it is economically feasible to do so.
- Developers shall separate submission of their segment and any bindings they create. The segment will be delivered to operational sites while the bindings will be distributed only to other developers.
- Developers should use Ada95 as the language of choice over earlier versions of Ada.

9.1.3 C/C++

This subsection contains requirements and suggestions that are specific to programming in C or C++.

- Developers should use ANSI C instead of Kernighan and Ritchie C because of the strong typing capabilities of ANSI C.
- Segments that have public APIs written in C shall support ANSI C function prototypes.
- Segments that have public APIs shall support linking with C++ modules. This is done by bracketing function definitions with

```
#ifdef __cplusplus
extern "C" {
#endif
```

function prototypes

```
#ifdef _cplusplus
}
#endif
```

- Segments written in C that have public APIs shall handle the condition where a header file is included twice. This is accomplished by bracketing the header file with `#ifndef` and `#endif` as follows:

```
#ifndef MYHEADER
#define MYHEADER

header file declarations

#endif
```

9.2 Development Directory Structure

Developers may use whatever directory structure is most appropriate for their development process. The installation tools will enforce the logical structure presented in Chapter 5. However, the COE development tools allow segments under development to be located arbitrarily on the disk. For example,

```
VerifySeg -p /home5/test/dev MySeg
```

indicates that the segment to be validated, `MySeg`, is located in the directory `/home5/test/dev`. Similarly,

```
TestInstall -p /home5/test/dev MySeg
```

allows the segment to be temporarily installed from this directory for testing and debugging.

Figure 9-1 shows an example segment directory structure. It has the advantage that it separates public and private code into different subdirectories. `MySeg/lib` contains public libraries provided by the segment, while `MySeg/include` contains public header (C/C++) or package definition (Ada) files. The `src/PrivLib` subdirectory should contain library modules that are private to the segment. Similarly, the subdirectory `src/PrivInclude` contains interface files that are private to the segment.

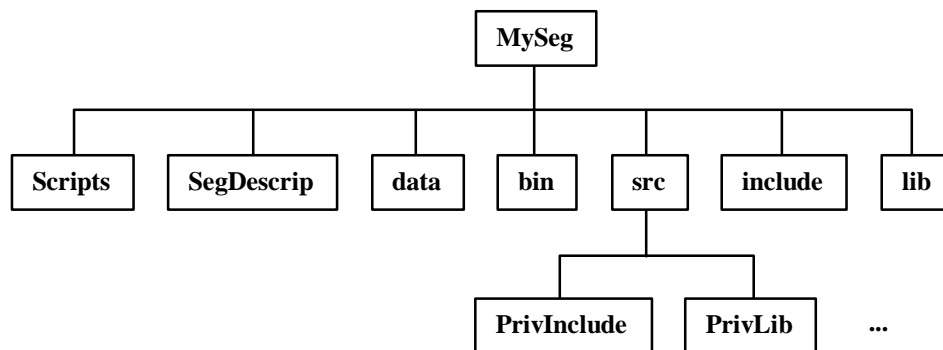


Figure 9-1: Example Development Directory Structure

This directory structure is not mandatory, except when source code is delivered to DISA; otherwise, it represents only one recommended approach. When source code is delivered to DISA, it shall be in the `src`, `include`, and `lib` directories as appropriate.

An advantage of structuring directories as shown in Figure 9-1 is that delivering software to other developers means that only one directory must be deleted: the `src` directory. Delivering the software to an operational site means that only three directories need to be deleted: `include`, `lib` (unless shared libraries are being used), and `src`. It is a simple matter to create automated scripts that can generate tapes for both types of deliveries. An additional benefit is that public and private files are separated in the directory structure for easier management and distribution.

9.3 Separating Out the Development Environment

The COE requires that a strict separation be maintained between the runtime environment and the development environment. This is true regardless of the target platform operating system (e.g., NT, Unix). For the NT¹ world, most development tools are structured in such a way that the development environment is self-contained in an integrated environment that is accessible from a GUI. For example, both Microsoft and Borland provide an integrated development environment for C++ that provides icon and menu access to compilers, linkers, editors, and other development tools. Both products provide a “directory browser” for identifying the location of source code and libraries, and the target directory for object code and executables. Moreover, they provide an interface for defining parameters such as compiler flags and preserve the settings and all other build-related information in a “project file.”

For Unix, however, integrated development environments are less common place. The next subsection describes an approach for preserving the separation of development and runtime Unix environments through the use of scripts. The concept is to put all runtime information into one script, and all development information in a separate script. While the approach between NT and Unix is considerably different, the COE stipulates a fundamental requirement to preserve a separation between the runtime and development environment. Developers shall preserve this separation regardless of the target operating system environment.

9.3.1 Unix Development Scripts

In the Unix environment, it is often convenient to locate development scripts in the same subdirectory as the runtime scripts (e.g., subdirectory `Scripts`). The recommended convention is to name development scripts with a `.dev` extension to distinguish them from runtime environment scripts. The `.runtime` extension can *not* be used since this has a special meaning within the COE as explained in Chapter 5.

Developers may define environment variables for locating source code directories, compilers, tools, and libraries. In addition, aliases can be defined as shortcuts for frequently executed commands. None of these examples are allowed in the runtime environment and hence must be placed in a development script such as `.cshrc.dev`.

9.3.2 NT² and Unix Recommendations

The following suggestions are made:

¹ The DII COE for NT is presently available only on PC platforms. Comments in this chapter should be understood in the context of Windows NT for PC-based platforms, even though the NT operating system is available on other commercial platforms. DII COE support for non-PC platforms is dependent upon requirements from the DII COE community.

² *ibid.*

- Define environment variables relative to *segprefix_HOME* where *segprefix* is the segment prefix. This allows segments to be easily relocated on the disk. (This suggestion is applicable to both Unix and NT.)
- Use environment variables to define where to place libraries and executables. (Unix only. For NT, use facilities provided by the development tools for locating libraries and executables.)
- Extend the path environment variable through concatenation - that is

```
set path = ($path $TOOLS)
```

where \$TOOLS is the location of the COE development tools (e.g., /h/TOOLS). (Unix only. For NT, use facilities provided by the development tools for locating tools.)

- Use the same script for all supported platforms through use of the environment variables *MACHINE_CPU* and *MACHINE_OS*. (Unix only. For NT, use facilities provided by the development tools for creating project files that allow multi-platform development support.)

9.3.3 Test Account Group

COE-component segment developers typically create servers that will be used by other segments in the operational system. However, the developers and the SSA need to be able to test the COE-component segments when there may not be available any mission-application segments, or even an account group segment, that will launch the servers and exercise the API interfaces.

To aid the SSA and other segment developers, it is recommended that COE-component segment developers create and deliver with the segment the following:

- **A test account group segment.** This segment should establish the environment that the COE segment is expected to run within, and contain details for how to correctly launch the services. This provides a way for the SSA to test the delivered segments, and it provides an example for system engineers and designers how the segment was intended to be used.
- **A “Run” script.** Chapter 5 indicates that account group segments must contain an executable that will launch the application. The test segment should also contain such an executable. This encapsulates in one place the information required to properly establish the runtime environment to launch the server, and it also identifies the sequence and command-line parameters, if any, required to launch the services.
- **Documentation.** The test segment and “Run” script should be documented to assist the system integrator, potential system designers, and the SSA.

The test segment and “Run” script should be packaged and delivered separately from the actual COE-component segment. This will ensure that the test segment does not inadvertently get delivered to an operational site, or get confused with account group segments that are intended to be part of the end system.

9.4 Private and Public Files

The software engineering principles of data abstraction and data hiding are important in designing segments. *Data abstraction* refers to the process of abstracting structures so that subscriber segments need not know low-level details of how data is physically organized. *Data hiding* refers to hiding data elements that subscriber segments do not need, or are not authorized, to directly access. Proper implementation of these two design principles prevents segments from affecting each other through inadvertent side effects and isolates one segment from changes in another.

It is also important to hide low-level functions and only provide access to segment functionality through a carefully controlled interface, the API. It is neither feasible nor desirable to make all functions in a segment available due to the sheer number of functions involved, and because changing a function that is being used directly by another developer may have significant impact.

These concepts are implemented in Ada through the *package* construct. C, however, does not contain an equivalent capability. The closest approximation in C is the *static* directive that makes a function visible only within the scope of the file containing the function definition. To compensate for structural inadequacies in C, developers must segregate software into public and private files, and into public and private directories. Since header files (e.g., .h files) are used to define the interface to C functions, the concept is that header files should be segregated into public and private files while public and private directories are used to provide the same concept for libraries. Moreover, segregation into distinct directories makes it easier to enforce the separation.

9.5 Developer's Toolkit

The Developer's Toolkit contains the components necessary for creating segments that use COE components. The toolkit does not need to be in segment format (it is not installed at operational sites), but it is a set of files and directories that may be downloaded electronically from the online library. Developer's may also contact the DII COE Configuration Management Department to receive the toolkit on magnetic media in relative "tar" format.

The Developer's Toolkit is distributed separately from the target COE-based system. However, components from the operational system (COE-component segments, shared libraries, etc.) are required for development. These may be obtained electronically from the online library, or on magnetic media from the DII COE Configuration Management Department. Classified or very large components will be distributed to developers via magnetic media. The toolkit does not duplicate any components available in the runtime system because this would create configuration management problems in ensuring that developers do not receive two different versions of the same module.

As distributed, the toolkit contains the following:

- API libraries and object code
- C header files for public APIs written in C
- Ada package definitions for APIs written in Ada
- Ada bindings for selected APIs
- API documentation in HTML format³
- API documentation in Unix man page format
- COE development tools (see Appendix C)
- Conventions for creating APIs

The toolkit does *not* contain any products that require a license (compilers, editors, RDBMS, etc.). It is the developer's responsibility to acquire these items as needed.

Developers may install the toolkit on the disk in whatever directories are desired. The standard location for toolkit components is:

C public header files	/h/COE/include
Ada public package definitions	/h/COE/include
public libraries	/h/COE/lib
executables	/h/TOOLS/bin
Unix man pages	/h/TOOLS/man
HTML documentation	/h/TOOLS/HTML

³ Documentation is delivered in only one format. The goal is to use HTML for programmer documentation because this is suitable for both NT and Unix platforms. However, some documentation is still in Unix man page format.

Certain tools from Appendix C are useful for both the development environment and the runtime environment. These tools are delivered with the operational system and are located under `/h/COE/bin`.

Developers should include `/h/TOOLS/bin` in the `path` environment variable for their development environment. `/h/TOOLS/man` should also be included in the search path for Unix man pages. The web browser should be set to find HTML documentation under `/h/TOOLS/HTML`.

Developers are encouraged to submit tools to the COE community for inclusion in the developer's toolkit. All tools submitted must be license and royalty free, and must include a man page for online documentation. Developers wishing to release source code for their contributed tools may do so, and the source code for the tool will be organized under the `/h/TOOLS/src` directory.

This page is intentionally blank.